Schedule

1. Introduction and Programming Basics (2-10-2014)
2. Performance analysis (6-10-2014)
3. Advanced CUDA programming (9-10-2014)
4. Case study: LOFAR telescope with many-cores
   by Rob van Nieuwpoort (??)
Before we start ...

- There were two assignments
  - Vectorization of complex-number division/multiplication
  - Compute the hardware performance for your (dream) machine

- No answers ... 😞
Complex numbers

- Two parts: real and imaginary
- Notation: $a+bi$ or $a.\text{re} + a.\text{im}*i$
  - $i = \sqrt{-1} \Rightarrow i^2 = -1$
- Multiplication:
  \[(a.\text{re}+a.\text{im}*i)*(b.\text{re}+b.\text{im}*i) = (a.\text{re}*b.\text{re} - a.\text{im}*b.\text{im}) + (a.\text{re}*b.\text{im}+a.\text{im}*b.\text{re})*i\]
- Division:
  \[(a.\text{re}+a.\text{im}*i) / (b.\text{re}+b.\text{im}*i) = (a.\text{re}*b.\text{re} + a.\text{im}*b.\text{im})/(b.\text{re}^2+b.\text{im}^2) + (a.\text{im}*b.\text{re} - a.\text{re}*b.\text{im})/(b.\text{re}^2+b.\text{im}^2)*i\]
Vectorization [1/2]

- **Addressing by position:**
  - `vec1.w, vec1.x, vec1.y, vec1.z = vec1[0],[1],[2],[3].`

- **Load/store:**
  - `vecA = load(A);`
  - `store(vecA, A);`

- **Addition, multiplication, subtraction, division:**
  - `vec3 = add/mul/sub/div(vec1, vec2);`

- **Shuffle:**
  - `Vec3 = shuffle(vec1, vec2, (a,b,c,d));`
    - `vec3.w = vec3[0] = vec1[a]; vec3.x = vec3[1] = vec1[b];`
for (j=0; j<N; j+=2) {
    vecA = load(A+2*j); vecB = load(B+2*j);
    // b.re*b.re, b.im*b.im
    vecT1=mul(vecB,vecB);
    // a.re*b.re, a.im*b.im
    vecT2=mul(vecA,vecB);
    // rotate A left and B right
    vecT3=shuffle(vecA,vecA,(1,2,3,0);
    vecT4=shuffle(vecB,vecB,(3,0,1,2);
    // b.re*a.im on positions 0 and 2
    vecT5=mul(vecB,vecT3);
    // b.im*a.re on positions 1 and 3
    vecT4=mul(vecA,vecT4);
    C[2*j]    = (vecT2.w+vecT2.x)/(vecT1.w+vecT2.x)
    C[2*j+1]  = ...
    C[2*j+2]  = ...
    C[2*j+3]  = ...
}
Today

- Introduction(/refresher) to CUDA
- Advanced programming
  - Synchronization and atomics
  - Memory coalescing
  - Shared Memory
  - Warps and Occupancy
  - Streams and Asynchronous Execution
  - Bank conflicts
  - Other features
- OpenCL brief introduction
NVIDIA GPUs
Reader:
- NVIDIA’s Next Generation CUDA Compute Architecture: Fermi

Also for the lab:
- NVIDIA’s CUDA Programmer Guide

Recommended further reading:

Note:
CUDA = Compute Unified Device Architecture
Fermi Streaming Multiprocessor

- 32 cores per SM (512 cores total)
- 64KB configurable L1 cache / shared memory
- 32,768 32-bit registers
Memory architecture (from Fermi)

- Configurable L1 cache per SM
  - 16KB L1 cache / 48KB Shared
  - 48KB L1 cache / 16KB Shared

- Shared L2 cache
Programming in CUDA
CUDA

- **CUDA: Scalable parallel programming**
  - C/C++ extensions

- Provide straightforward mapping onto hardware
  - Good fit to GPU architecture
  - Maps well to multi-core CPUs too

- Scale to 1000s of cores & 100,000s of threads
  - GPU threads are lightweight — create / switch is free
  - GPU needs 1000s of threads for full utilization
Parallel Abstractions in CUDA

- Hierarchy of (MANY) concurrent threads
  - Scheduling
  - Memory accesses

- Synchronization
  - Lightweight primitives
  - Options and lack of options

- Cooperation
  - Shared memory
Hierarchy of concurrent threads

- Parallel kernels composed of many threads
  - All threads execute the same sequential program
  - Called the kernel

- Threads are grouped into thread blocks
  - Threads in the same block can cooperate
  - Threads in different blocks cannot!

- All thread blocks are organized in a Grid

- Threads/blocks have unique IDs
Grids, Thread Blocks and Threads
CUDA virtualizes the physical hardware

- A block is a virtualized streaming multiprocessor (threads, shared memory)
- A thread is a virtualized scalar processor (registers, PC, state)

Scheduled onto physical hardware without pre-emption

- threads/blocks launch & run to completion
- blocks should be independent
Memory Scopes

- **Per-thread private memory**
  - Each thread has its own local memory
    - Stacks, other private data
  - Registers

- **Per-SM shared memory**
  - Small memory close to the processor, low latency

- **Device memory**
  - GPU frame buffer
  - Can be accessed by any thread in any SM
## CUDA Variable Type Qualifiers

<table>
<thead>
<tr>
<th>Variable declaration</th>
<th>Memory</th>
<th>Scope</th>
<th>Lifetime</th>
</tr>
</thead>
<tbody>
<tr>
<td>int var;</td>
<td>register</td>
<td>thread</td>
<td>thread</td>
</tr>
<tr>
<td>int array_var[10];</td>
<td>local</td>
<td>thread</td>
<td>thread</td>
</tr>
<tr>
<td><strong>shared</strong> int shared_var;</td>
<td>shared</td>
<td>block</td>
<td>block</td>
</tr>
<tr>
<td><strong>device</strong> int global_var;</td>
<td>device</td>
<td>grid</td>
<td>application</td>
</tr>
<tr>
<td><strong>constant</strong> int constant_var;</td>
<td>constant</td>
<td>grid</td>
<td>application</td>
</tr>
</tbody>
</table>
Memory Spaces in CUDA

- **Host**
  - Private data
  - Shared data (per block)
  - Global data

- **Device Memory**
  - Constant Memory
  - Texture Memory
Device Memory

- CPU and GPU have separate memory spaces
  - Data is moved across PCI-e bus
  - Use functions to allocate/set/copy memory on GPU
- Pointers are just addresses
  - Can’t tell from the pointer value whether the address is on CPU or GPU
  - Must exercise care when dereferencing:
    - Dereferencing CPU pointer on GPU will likely crash
    - Same for vice versa
Other memories

- **Constant memory** – *ReadOnly*
  - Data resides in device memory
  - Manually managed
  - Small (e.g., 64KB)
  - Use when all threads in a block read the same address
    - Serializes otherwise

- **Textures*** – *ReadOnly*
  - Dimension restricted by hardware
  - Data resides in device memory
    - Different read path, includes specialized caches
    - Specialized spatial locality
  - Dedicated API for reading and processing
    - Normalized (float) coordinates
    - Interpolated values
    - Specified out-of-bounds behavior

ECC (Error-Correcting Code)

- All major internal memories are ECC protected
  - Register file, L1 cache, L2 cache
- DRAM protected by ECC (on Tesla only)
- ECC is a must have for many computing applications
CUDA: Synchronization and Atomics
Thread Scheduling

- Order of threads within a block is undefined!
  - Threads are grouped in warps (32)
    - AMD calls it “a wavefront” (64)

- Order in which thread blocks are mapped and scheduled is undefined!
  - Blocks run to completion on one SM without preemption
  - Can run in any order
    - Any possible interleaving of blocks should be valid
  - Can run concurrently OR sequentially
Global synchronization

- We launch many more blocks than physical SM’s.
- Each block might/should have more threads than the SM’s cores

```c
__global__ void my_kernel() {
    step1; // compute some values in a global array
    // wait for *all* threads to finish
    __my_global_barrier();
    step2; // use the array
}

int main() {
    dim3 blockSize(32, 32);
    dim3 gridSize(100, 100, 100);
    my_kernel<<<gridDim, blockDim>>>();
}
```
Global synchronization

- Q: How do we do global synchronization with these scheduling semantics?
  - NOT POSSIBLE on the device itself
  - POSSIBLE by finishing a grid, and starting a new one!
    - Split work in two different kernels

```c
step1<<<grid1,blk1>>>(...);
// CUDA ensures that all writes from step1 are complete.
step2<<<grid2,blk2>>>(...);
```

- Global, constant, and texture memories are persistent!
  - We don’t need extra data copying.
Memory consistency

- Device (global) memory is not coherent!
  - No insurance that different
- Share data between streaming multiprocessors
  - Potential write hazards!
- Use **atomics** to allow consistency for global (and shared memory) variables!
- Evolution:
  - Fermi has reasonable atomics for both shared and global memory
  - Kepler increases *global memory atomics* performance vs. Fermi
  - Maxwell uses native support for shared memory atomics
    - Much faster than Fermi and Kepler
Atomics

- Guarantee that only a single thread has access to a piece of memory during an operation
  - Ordering is still arbitrary
- Different types of atomic instructions
  - Atomic Add, Sub, Exch, Min, Max, Inc, Dec, CAS, And, Or, Xor
- Both device memory and shared memory
// Determine frequency of colors in a picture.
// Colors have already been converted into integers
// between 0 and 255.
// Each thread looks at one pixel,
// and increments a counter

__global__ void histogram(int* colors, int* buckets)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    int c = colors[i];
    buckets[c] += 1;
}
Example: Histogram

// Determine frequency of colors in a picture.
// Colors have already been converted into integers
// between 0 and 255.
// Each thread looks at one pixel,
// and increments a counter

__global__ void histogram(int* colors, int* buckets)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    int c = colors[i];
    buckets[c] += 1;
}
Example: Histogram

// Determine frequency of colors in a picture.
// Colors have already been converted into integers
// between 0 and 255.
// Each thread looks at one pixel,
// and increments a counter atomically

__global__ void histogram(int* colors, int* buckets)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    int c = colors[i];
    atomicAdd(&buckets[c], 1);
}
CUDA: Memory coalescing
Coalescing

traditional multi-core
optimal memory access pattern

<table>
<thead>
<tr>
<th>thread 0</th>
<th>address 0</th>
<th>address 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>t = 0</td>
<td>address 0</td>
<td>address 1</td>
</tr>
<tr>
<td>t = 1</td>
<td>address 1</td>
<td>address 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>thread 1</th>
<th>address 2</th>
<th>address 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>t = 0</td>
<td>address 2</td>
<td>address 3</td>
</tr>
<tr>
<td>t = 1</td>
<td>address 3</td>
<td>address 4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>thread 2</th>
<th>address 4</th>
<th>address 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>t = 0</td>
<td>address 4</td>
<td>address 5</td>
</tr>
<tr>
<td>t = 1</td>
<td>address 5</td>
<td>address 6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>thread 3</th>
<th>address 6</th>
<th>address 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>t = 0</td>
<td>address 6</td>
<td>address 7</td>
</tr>
<tr>
<td>t = 1</td>
<td>address 7</td>
<td></td>
</tr>
</tbody>
</table>

many-core GPU
optimal memory access pattern

<table>
<thead>
<tr>
<th>thread 0</th>
<th>address 0</th>
<th>address 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>t = 0</td>
<td>address 0</td>
<td>address 1</td>
</tr>
<tr>
<td>t = 1</td>
<td>address 1</td>
<td>address 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>thread 1</th>
<th>address 2</th>
<th>address 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>t = 0</td>
<td>address 2</td>
<td>address 3</td>
</tr>
<tr>
<td>t = 1</td>
<td>address 3</td>
<td>address 4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>thread 2</th>
<th>address 4</th>
<th>address 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>t = 0</td>
<td>address 4</td>
<td>address 5</td>
</tr>
<tr>
<td>t = 1</td>
<td>address 5</td>
<td>address 6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>thread 3</th>
<th>address 6</th>
<th>address 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>t = 0</td>
<td>address 6</td>
<td>address 7</td>
</tr>
<tr>
<td>t = 1</td>
<td>address 7</td>
<td></td>
</tr>
</tbody>
</table>
Memory coalescing refers to combining multiple memory accesses into a single transaction.
Consider the stride of your accesses

```c
__global__ void foo(int* input, float3* input2) {

    int i = blockDim.x * blockIdx.x + threadIdx.x;

    // Stride 1, full bandwidth used!
    int a = input[i];

    // Stride 2, 50% of the bandwidth is wasted
    int b = input[2*i];

    // Stride 3, 67% of the bandwidth is wasted
    float c = input2[i].x;
}
```
Example: Array of Structures (AoS)

Struct record {
    int key;
    int value;
    int flag;
};

record *d_AoS_data;
cudaMalloc((void**)&d_AoS_data, ...);

kernel {
    threadID = blockDim.x * blockIdx.x + threadIdx.x;
    // ...
    d_AoS_data[threadID].value += i; // wastes bandwidth!
    // ...
}
Example: Structure of Arrays (SoA)

Struct SoA {
    int* keys;
    int* values;
    int* flags;
};

SoA d_SoA_data;
cudaMalloc((void**)&d_SoA_data.keys, ...);
cudaMalloc((void**)&d_SoA_data.values, ...);
cudaMalloc((void**)&d_SoA_data.flags, ...);

kernel {
    threadID = blockDim.x * blockIdx.x + threadIdx.x;
    ...
    d_SoA_data.values[threadID] += i; // full bandwidth!
    ...
}
__global__ void bar(record* AoS_data, SoA SoA_data) {
  int i = blockDim.x * blockIdx.x + threadIdx.x;

  // AoS wastes bandwidth
  int key1 = AoS_data[i].key;

  // SoA efficient use of bandwidth
  int key2 = SoA_data.keys[i];
}
Memory Coalescing

- Structure of arrays is often better than array of structures
- Stride 1 access patterns are preferred!
  - Other patterns can still get benefits
- Unpredictable/irregular access patterns
  - Case-by-case performance impact

- No coalescing => performance loss 10 – 30x!
CUDA: Using Shared Memory
Using shared memory

- Equivalent with providing software caching
  - **Explicit**: Load data to be re-used in shared memory
  - Use it for computation
  - **Explicit**: Store results back to global memory

- All threads in a block share memory
  - Load/Store: using all threads
  - **Barrier**: `__syncthreads`
    - Guard against using uninitialized data – not all threads have finished loading data to shared memory
    - Guard against corrupting live data – not all threads have finished computing
A Common Programming Strategy

- Partition data into subsets that fit into shared memory
A Common Programming Strategy

- Handle each data subset with one thread block
A Common Programming Strategy

- Load the subset from device memory to shared memory, using multiple threads to exploit memory-level parallelism
- Perform the computation on the subset from shared memory
A Common Programming Strategy

- Copy the result from shared memory back to device memory
Caches vs. Shared Memory

- Since Fermi, NVIDIA GPUs feature BOTH hardware **L1 caches** and **shared memory** per SM
  - They share the same space
    - $\frac{3}{4}$ Cache + $\frac{1}{4}$ Shared Memory  OR
    - $\frac{1}{4}$ Cache + $\frac{3}{4}$ Shared Memory

- **L1 Cache**
  - Hardware caching enabled
    - The HW decides what goes in or out and when

- **Shared memory**
  - Software manages what goes in/out
  - Allows more complex access patterns to be cached
Matrix multiplication example

- \( C = A \times B \)
  - \( C(i,j) = \text{sum}(\text{dot(row}(A,i),\text{col}(B,j))) \)

- Parallelization strategy
  - Each thread computes one \( C \) element
  - 2D kernel
Matrix multiplication implementation

```c
__global__ void mat_mul(float *a, float *b,
                        float *c, int width)
{
    // calc row & column index of output element
    int row = blockIdx.y*blockDim.y + threadIdx.y;
    int col = blockIdx.x*blockDim.x + threadIdx.x;

    float result = 0;

    // do dot product between row of a and column of b
    for(int k = 0; k < width; k++) {
        result += a[row*width+k] * b[k*width+col];
    }
    c[row*width+col] = result;
}
```
## Matrix multiplication performance

<table>
<thead>
<tr>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loads per dot product term</td>
<td>$2 \ (a \ and \ b) = 8 \text{ bytes}$</td>
</tr>
<tr>
<td>FLOPS</td>
<td>$2 \ (\text{multiply and add})$</td>
</tr>
<tr>
<td>AI</td>
<td>$\frac{2}{8} = 0.25$</td>
</tr>
<tr>
<td>Performance GTX 580</td>
<td>1581 \text{ GFLOPs}</td>
</tr>
<tr>
<td>Memory bandwidth GTX 580</td>
<td>192 \text{ GB/s}</td>
</tr>
<tr>
<td>Attainable performance</td>
<td>$192 \times 0.25 = 48 \text{ GFLOPS}$</td>
</tr>
<tr>
<td>Maximum efficiency</td>
<td>3.0 % of theoretical peak</td>
</tr>
</tbody>
</table>
Data reuse

- Each input element in A and B is read WIDTH times

IDEA:
- Load elements into shared memory
- Have several threads use local version to improve memory bandwidth
Using shared memory

- Partition kernel loop into phases
- In each thread block, load a tile of both matrices into shared memory each phase
- Each phase, each thread computes a partial result
Matrix multiply with shared memory

```c
__global__ void mat_mul(float *a, float *b, float *c, int width) {

    // shorthand
    int tx = threadIdx.x, ty = threadIdx.y;
    int bx = blockIdx.x, by = blockIdx.y;

    // allocate tiles in shared memory
    __shared__ float s_a[TILE_WIDTH][TILE_WIDTH];
    __shared__ float s_b[TILE_WIDTH][TILE_WIDTH];

    // calculate the row & column index from A,B
    int row = by*blockDim.y + ty;
    int col = bx*blockDim.x + tx;

    float result = 0;
}
```
Matrix multiply with shared memory

// loop over input tiles in phases, p = crt. phase
for(int p = 0; p < width/TILE_WIDTH; p++) {
    // collaboratively load tiles into shared memory
    s_a[ty][tx] = a[row*width + (p*TILE_WIDTH + tx)];
    s_b[ty][tx] = b[(p*TILE_WIDTH + ty)*width + col];
    // barrier: ALL writes to shared memory finished
    __syncthreads();

    // dot product between row of s_a and col of s_b
    for(int k = 0; k < TILE_WIDTH; k++) {
        result += s_a[ty][k] * s_b[k][tx];
    }
    // barrier: ALL reads of shared memory finished
    __syncthreads();
}

c[row*width+col] = result;
Use of Barriers in \texttt{mat\_mul}

- Two barriers per phase:
  - \texttt{__syncthreads} after all data is loaded into shared memory
  - \texttt{__syncthreads} after all data is read from shared memory
    - Second \texttt{__syncthreads} in phase \( p \) guards the load in phase \( p+1 \)

- Formally, \texttt{__syncthreads} is a barrier for shared memory for a block of threads:

  ```
  "void \texttt{__syncthreads}();
  waits until all threads in the thread block have reached this point \textbf{and} all global and shared memory accesses made by these threads prior to \texttt{__syncthreads}() are visible to all threads in the block."
  ```
## Matrix multiplication performance

<table>
<thead>
<tr>
<th></th>
<th>Original</th>
<th>shared memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global loads</td>
<td>$2N^3 \times 4$ bytes</td>
<td>$(2N^3 / \text{TILE_WIDTH}) \times 4$ bytes</td>
</tr>
<tr>
<td>Total ops</td>
<td>$2N^3$</td>
<td>$2N^3$</td>
</tr>
<tr>
<td>AI</td>
<td>0.25</td>
<td>$0.25 \times \text{TILE_WIDTH}$</td>
</tr>
</tbody>
</table>

### Performance GTX 580
- 1581 GFLOPs

### Memory bandwidth GTX 580
- 192 GB/s

### AI needed for peak
- $1581 / 192 = 8.23$

### TILE\_WIDTH required to achieve peak
- $0.25 \times \text{TILE\_WIDTH} = 8.23$
- $\text{TILE\_WIDTH} = 32.9$
CUDA: Warps and Occupancy
SMs implement zero-overhead warp scheduling
- A warp is a group of 32 threads that runs concurrently on an SM
- At any time, the number of warps concurrently executed by an SM is limited by its number of cores.
- Warps whose next instruction has its inputs ready for consumption are eligible for execution
- Eligible Warps are selected for execution on a prioritized scheduling policy
- All threads in a warp execute the same instruction when selected

Instruction: 1 2 3 4 5 6 1 2 1 2 3 4 7 8 1 2 1 2 3 4

TB = Thread Block, W = Warp
Stalling warps

- What happens if all warps are stalled?
  - No instruction issued → performance lost

- Most common reason for stalling?
  - Waiting on global memory

- If your code reads global memory every couple of instructions
  - You should try to maximize occupancy
What determines occupancy?

Limited resources!
- Register usage per thread
- Shared memory per thread block
- Pool of registers and shared memory per SM
  - Each thread block grabs registers & shared memory
  - If one or the other is fully utilized ➞ no more thread blocks
Resource Limits (2)

- Can only have $P$ thread blocks per SM
  - If they’re too small, can’t fill up the SM
  - Need 128 threads / block on gt200 (4 cycles/instruction)
  - Need 192 threads / block on Fermi (6 cycles/instruction)

- Higher occupancy has diminishing returns for hiding latency
Hiding Latency with more threads

Throughput, 32-bit words

<table>
<thead>
<tr>
<th>Threads Per Multiprocessor</th>
<th>GB/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>128</td>
<td>10</td>
</tr>
<tr>
<td>256</td>
<td>20</td>
</tr>
<tr>
<td>384</td>
<td>30</td>
</tr>
<tr>
<td>512</td>
<td>40</td>
</tr>
<tr>
<td>640</td>
<td>50</td>
</tr>
<tr>
<td>768</td>
<td>60</td>
</tr>
<tr>
<td>896</td>
<td>70</td>
</tr>
<tr>
<td>1024</td>
<td>80</td>
</tr>
</tbody>
</table>
How do you know what you’re using?

- Use compiler flags to get register and shared memory usage
  - “nvcc -Xptxas -v”

- Use the NVIDIA Profiler

- Plug those numbers into CUDA Occupancy Calculator

- Maximize occupancy for improved performance
  - Empirical rule! Don’t overuse!
1. Select Compute Capability (click): 1.3

2. Enter your resource usage:
   - Threads Per Block: 128
   - Registers Per Thread: 28
   - Shared Memory Per Block (bytes): 640

3. GPU Occupancy Data is displayed here and in the graphs:
   - Active Threads per Multiprocessor: 512
   - Active Warps per Multiprocessor: 16
   - Active Thread Blocks per Multiprocessor: 4
   - Occupancy of each Multiprocessor: 50%

4. Physical Limits for GPU Compute Capability: 1.3
   - Threads per Warp: 32
   - Warps per Multiprocessor: 32
   - Threads per Multiprocessor: 1024
   - Thread Blocks per Multiprocessor: 8
   - Total # of 32-bit registers per Multiprocessor: 16384
   - Register allocation unit size: 512
   - Register allocation granularity: block
   - Shared Memory per Multiprocessor (bytes): 16384
   - Shared Memory Allocation unit size: 512
   - Warp allocation granularity (for register allocation): 2

5. Allocation Per Thread Block
   - Warps: 4
   - Registers: 3684
   - Shared Memory: 1024
   - These data are used in computing the occupancy data in blue

6. Maximum Thread Blocks Per Multiprocessor
   - Blocks Limited by Max Warps/Blocks per Multiprocessor: 8
   - Limited by Registers per Multiprocessor: 16
   - Limited by Shared Memory per Multiprocessor: 16
   - Thread Block Limit Per Multiprocessor highlighted: RED

7. CUDA Occupancy Calculator
   - Version: 2.0

The other data points represent the range of possible block sizes, register counts, and shared memory allocation.
Thread divergence

- “I heard GPU branching is expensive. Is this true?”

```c
__global__ void Divergence(float* dst, float* src)
{
    float value = 0.0f;

    if (threadIdx.x % 2 == 0)
        // active threads : 50%
        value = src[0] + 5.0f;
    else
        // active threads : 50%
        value = src[0] - 5.0f;

    dst[index] = value;
}
```
Execution

Worst case performance loss: 50% compared with the non divergent case.

```c
unsigned int index = ( blockDim.x * blockIdx.x ) + threadIdx.x;
float value = 0.0f;

if ( threadIdx.x % 2 == 0 )

value = PathA( src );

value = PathB( src );

dst[index] = value;
```
Another example

Time (clocks)

ALU 1  ALU 2  ...  ...  ALU 8

1  2  ...  8

Not all ALUs do useful work!
Worst case: 1/8 peak performance

(assume logic below is to be executed for each element in input array 'A', producing output into the array 'result')

<unconditional code>

float x = A[i];
if (x > 0) {
    float tmp = exp(x, 5.f);
    tmp *= kMyConst1;
    x = tmp + kMyConst2;
} else {
    float tmp = kMyConst1;
    x = 2.f * tmp;
}

<resume unconditional code>

result[i] = x;
Performance penalty?

- Depends on the amount of divergence
  - Worst case: 1/32 performance
    - When each thread does something different

- Depends on whether branching is data- or ID-dependent
  - If ID – consider grouping threads differently
  - If data – consider sorting

- Non-diverging warps => NO performance penalty
  - In this case, branches are not expensive …
CUDA: Streams
What are streams?

- Stream = a sequence of operations that execute on the device in the order in which they are issued by the host code.
- Same stream: In-Order execution
- Different streams: Out-of-Order execution

- Default stream = Synchronizing stream
  - No operation in the default stream can begin until all previously issued operations in any stream on the device have completed.
  - An operation in the default stream must complete before any other operation in any stream on the device can begin.
Default stream: example

cudaMemcpy(d_a, a, numBytes, cudaMemcpyHostToDevice);
increment<<<1,N>>>(d_a);
CpuFunction(b);
cudaMemcpy(a, d_a, numBytes, cudaMemcpyDeviceToHost);

- All operations happen in the same stream
- Device (GPU)
  - Synchronous execution
    - all operations execute (in order), one after the previous has finished
  - Unaware of CpuFunction()
- Host (CPU)
  - Launches increment and regains control
  - *May* execute CpuFunction *before* increment has finished
  - Final copy starts *after* both increment and CpuFunction() have finished
Non-default streams

- Enable asynchronous execution and overlaps
  - Require special creation/deletion of streams
    - `cudaStreamCreate(&stream1)`
    - `cudaStreamDestroy(stream1)`
  - Special memory operations
    - `cudaMemcpyAsync(deviceMem, hostMem, size, cudaMemcpyHostToDevice, stream1)`
  - Special kernel parameter (the 4th one)
    - `increment<1, N, 0, stream1>>(d_a)`

- Synchronization
  - All streams
    - `cudaDeviceSynchronize()`
  - Specific stream:
    - `cudaStreamSynchronize(stream1)`
Computation vs. communication

//Single stream, numBytes = 16M, numElements = 4M
cudaMemcpy(d_a, a, numBytes, cudaMemcpyHostToDevice);
kernel<<blocks,threads>>>(d_a, firstElement);
cudaMemcpy(a, d_a, numBytes, cudaMemcpyDeviceToHost);

C1060 (pre-Fermi): 12.9ms

C2050 (Fermi): 9.9ms
for (int i = 0; i < nStreams; ++i) {
    int offset = i * streamSize;
    cudaMemcpyAsync(&d_a[offset], &a[offset], streamBytes, stream[i]);
    kernel<<<blocks, threads, 0, stream[i]>>>(d_a, offset);
    cudaMemcpyAsync(&a[offset], &d_a[offset], streamBytes, stream[i]);
}

C1060 (pre-Fermi): 13.63 ms (worse than sequential)

C2050 (Fermi): 5.73 ms (better than sequential)
for (int i = 0; i < nStreams; ++i) offset[i]=i * streamSize;
for (int i = 0; i < nStreams; ++i)
cudaMemcpyAsync(&d_a[offset[i]], &a[offset[i]], streamBytes, 
cudaMemcpyHostToDevice, stream[i]);

for (int i = 0; i < nStreams; ++i)
    kernel<<blocks,threads,0,stream[i]>>(d_a, offset);

for (int i = 0; i < nStreams; ++i)
cudaMemcpyAsync(&a[offset], &d_a[offset], streamBytes, 
cudaMemcpyDeviceToHost, stream[i]);

C1060 (pre-Fermi): 8.84 ms (better than sequential)

C2050 (Fermi): 7.59 ms (better than sequential, worse than v1)

CUDA: Shared memory bank conflicts
- Shared memory is banked
  - Only matters for threads within a warp
  - Full performance with some restrictions
    - Threads can each access different banks
    - Or can all access the same value
- Consecutive words are in different banks
- If two or more threads access the same bank but different value, we get bank conflicts
Bank Addressing Examples: OK

- **No Bank Conflicts**

  - Thread 0
  - Thread 1
  - Thread 2
  - Thread 3
  - Thread 4
  - Thread 5
  - Thread 6
  - Thread 7
  - Thread 15

  - Bank 0
  - Bank 1
  - Bank 2
  - Bank 3
  - Bank 4
  - Bank 5
  - Bank 6
  - Bank 7
  - Bank 15

- **No Bank Conflicts**

  - Thread 0
  - Thread 1
  - Thread 2
  - Thread 3
  - Thread 4
  - Thread 5
  - Thread 6
  - Thread 7
  - Thread 15

  - Bank 0
  - Bank 1
  - Bank 2
  - Bank 3
  - Bank 4
  - Bank 5
  - Bank 6
  - Bank 7
  - Bank 15
Bank Addressing Examples: BAD

- **2-way Bank Conflicts**
  - Thread 0
  - Thread 1
  - Thread 2
  - Thread 3
  - Thread 4
  - Thread 8
  - Thread 9
  - Thread 10
  - Thread 11
  - Bank 0
  - Bank 1
  - Bank 2
  - Bank 3
  - Bank 4
  - Bank 5
  - Bank 6
  - Bank 7
  - Bank 15

- **8-way Bank Conflicts**
  - Thread 0
  - Thread 1
  - Thread 2
  - Thread 3
  - Thread 4
  - Thread 5
  - Thread 6
  - Thread 7
  - Thread 15
  - Bank 0
  - Bank 1
  - Bank 2
  - Bank 3
  - Bank 4
  - Bank 5
  - Bank 6
  - Bank 7
  - Bank 8
  - Bank 9
  - Bank 15
Trick to Assess Performance Impact

- Change all shared memory reads to the same value
- All broadcasts = no conflicts
- Will show how much performance could be improved by eliminating bank conflicts

- The same doesn’t work for shared memory writes
  - So, replace shared memory array indices with `threadIdx.x`
  - (Could also be done for the reads)
CUDA: Many other features
HyperQ

CPU Cores Simultaneously Run Tasks on Kepler

FERMI
1 MPI Task at a Time

KEPLER
32 Simultaneous MPI Tasks
Dynamic parallelism

Dynamic Parallelism

GPU Adapts to Data, Dynamically Launches New Threads

CPU

Fermi GPU

CPU

Kepler GPU
Dynamic parallelism

**Fermi Workflow**

Stream Queue
Ordered queues of grids

One-way Flow

Work Distributor
Tracks blocks issued from grids

15 Active Grids

**Kepler Workflow**

Stream Queues
Ordered queues of grids

CUDA-Created Work

Grid Management Unit
Pending & suspended grids
1000's of pending grids

Two-way link allows pausing dispatch

Work Distributor
Actively dispatching grids

32 Active Grids

SM
SM
SM
SM

SMX
SMX
SMX
SMX
GPUDirect

Direct Transfers between GPU and 3rd Party Devices
In search of portability

OpenCL
Portability

- Inter-family vs inter-vendor
  - NVIDIA Cuda runs on all NVIDIA GPU families
  - OpenCL runs on all GPUs, Cell, CPUs

- Parallelism portability
  - Different architecture requires different granularity
  - Task vs data parallel

- Performance portability
  - Can we express platform-specific optimizations?
The Khronos group

Over 100 companies creating visual computing standards

Board of Promoters
OpenCL: Open Compute Language

- Architecture independent
- Explicit support for many-cores
- Low-level host API
  - Uses C library, no language extensions
- Separate high-level kernel language
  - Explicit support for vectorization
- Run-time compilation
- Architecture-dependent optimizations
  - Still needed
  - Possible
## Cuda vs OpenCL Terminology

<table>
<thead>
<tr>
<th>CUDA</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread</td>
<td>Work item</td>
</tr>
<tr>
<td>Thread block</td>
<td>Work group</td>
</tr>
<tr>
<td>Device memory</td>
<td>Global memory</td>
</tr>
<tr>
<td>Constant memory</td>
<td>Constant memory</td>
</tr>
<tr>
<td>Shared memory</td>
<td>Local memory</td>
</tr>
<tr>
<td>Local memory</td>
<td>Private memory</td>
</tr>
</tbody>
</table>
## Cuda vs OpenCL Qualifiers

### Functions

<table>
<thead>
<tr>
<th>CUDA</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>global</strong></td>
<td>__kernel</td>
</tr>
<tr>
<td><strong>device</strong></td>
<td>(no qualifier needed)</td>
</tr>
</tbody>
</table>

### Variables

<table>
<thead>
<tr>
<th>CUDA</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>constant</strong></td>
<td>__constant</td>
</tr>
<tr>
<td><strong>device</strong></td>
<td>__global</td>
</tr>
<tr>
<td><strong>shared</strong></td>
<td>__local</td>
</tr>
</tbody>
</table>
## Cuda vs OpenCL Indexing

<table>
<thead>
<tr>
<th>CUDA</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>gridDim</td>
<td>get_num_groups()</td>
</tr>
<tr>
<td>blockDim</td>
<td>get_local_size()</td>
</tr>
<tr>
<td>blockIdx</td>
<td>get_group_id()</td>
</tr>
<tr>
<td>threadIdx</td>
<td>get_local_id()</td>
</tr>
<tr>
<td>Calculate manually</td>
<td>get_global_id()</td>
</tr>
<tr>
<td>Calculate manually</td>
<td>get_global_size()</td>
</tr>
</tbody>
</table>

```c
__syncthreads() \rightarrow \text{barrier()}
```
Vector add: Cuda vs OpenCL kernel

CUDA

```c
__global__ void vectorAdd(float* a, float* b, float* c) {
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    c[index] = a[index] + b[index];
}
```

OpenCL

```c
__kernel void vectorAdd(__global float* a, __global float* b, __global float* c) {
    int index = get_global_id(0);
    c[index] = a[index] + b[index];
}
```
const size_t workGroupSize = 256;
const size_t nrWorkGroups = 3;
const size_t totalSize = nrWorkGroups * workGroupSize;

cl_platform_id platform;
clGetPlatformIDs(1, &platform, NULL);

// create properties list of key/values, 0-terminated.
cl_context_properties props[] = {
    CL_CONTEXT_PLATFORM, (cl_context_properties)platform, 0
};

cl_context context = clCreateContextFromType(props,
    CL_DEVICE_TYPE_GPU, 0, 0, 0);
cl_device_id device;
clGetDeviceIDs(platform, CL_DEVICE_TYPE_DEFAULT, 1, &device, NULL);

// create command queue on 1st device the context reported
cl_command_queue commandQueue =
    clCreateCommandQueue(context, device, 0, 0);

// create & compile program
cl_program program = clCreateProgramWithSource(context, 1, &programSource, 0, 0);
clBuildProgram(program, 0, 0, 0, 0, 0, 0, 0);

// create kernel
cl_kernel kernel = clCreateKernel(program, "vectorAdd", 0);
float* A, B, C = new float[totalSize]; // alloc host vecs
// initialize host memory here...

// allocate device memory
cl_mem deviceA = clCreateBuffer(context,
    CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
    totalSize * sizeof(cl_float), A, 0);

cl_mem deviceB = clCreateBuffer(context,
    CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
    totalSize * sizeof(cl_float), B, 0);

cl_mem deviceC = clCreateBuffer(context,
    CL_MEM_WRITE_ONLY, totalSize * sizeof(cl_float), 0, 0);
// setup parameter values
clSetKernelArg(kernel, 0, sizeof(cl_mem), &deviceA);
clSetKernelArg(kernel, 1, sizeof(cl_mem), &deviceB);
clSetKernelArg(kernel, 2, sizeof(cl_mem), &deviceC);

cEnqueueNDRangeKernel(commandQueue, kernel, 1, 0,
   &totalSize, &workGroupSize, 0,0,0); // execute kernel

// copy results from device back to host, blocking
cEnqueueReadBuffer(commandQueue, deviceC, CL_TRUE, 0,
   totalSize * sizeof(cl_float), C, 0, 0, 0);

delete[] A, B, C; // cleanup
cReleaseMemObject(deviceA); clReleaseMemObject(deviceB);
cReleaseMemObject(deviceC);
Summary and Conclusions
Summary and conclusions

- Higher performance cannot be reached by increasing clock frequencies anymore
- Solution: introduction of large-scale parallelism
  - Many-cores are here to stay
- Multiple cores on a chip
  - Today:
    - Up to 61 CPU cores in a node
    - Up to 3200 cores on a single GPU
  - Host system can contain multiple GPUs: 10,000+ cores
  - We can build clusters of these nodes!
- Future: 100,000s – millions of cores?
Many different types of many-core hardware

Very different properties
- Performance
- Programmability

The memory is the main bottleneck of all these platforms
- Different memory spaces pose different challenges

Performance analysis and estimation
- Amdahl’s Law
- Arithmetic intensity / Operational intensity
- Roofline model
Many different many-core programming models

Most models are hardware-induced, require low-level optimizations:
- Vectorization
- Coalescing
- Explicit software caching (shared memory on GPU)

Future
- Cuda? OpenCL?
- OpenACC?
Backup slides
Example: Work queue

// For algorithms where the amount of work per item 
// is highly non-uniform, it often makes sense to 
// continuously grab work from a queue.

__global__
void workq(int* work_q, int* q_counter,
            int queue_max, int* output)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    int q_index = atomicInc(q_counter, queue_max);
    int result = do_work(work_q[q_index]);
    output[q_index] = result;
}
// Adjacent Difference application:
// compute result[i] = input[i] - input[i-1]

__global__ void adj_diff_naive(int *result, int *input) {
    // compute this thread's global index
    unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;

    if(i > 0) {
        // each thread loads two elements from device memory
        int x_i = input[i];
        int x_i_minus_one = input[i-1];

        result[i] = x_i - x_i_minus_one;
    }
}
// Adjacent Difference application:
// compute result[i] = input[i] - input[i-1]

__global__ void adj_diff_naive(int *result, int *input) {
    unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;

    if(i > 0) {
        int x_i = input[i];
        int x_i_minus_one = input[i-1];

        result[i] = x_i - x_i_minus_one;
    }
}
// Adjacent Difference application:
// compute result[i] = input[i] – input[i-1]

__global__ void adj_diff_naive(int *result, int *input) {
    // compute this thread’s global index
    unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;

    if(i > 0) {
        // each thread loads two elements from device memory
        int x_i = input[i];
        int x_i_minus_one = input[i-1];

        result[i] = x_i - x_i_minus_one;
    }
}

The next thread also reads input[i]
__global__ void adj_diff(int *result, int *input) {
    unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;

    __shared__ int s_data[BLOCK_SIZE]; // shared, 1 elt / thread
    // each thread reads 1 device memory elt, stores it in s_data
    s_data[threadIdx.x] = input[i];

    // avoid race condition: ensure all loads are complete
    __syncthreads();

    if(threadIdx.x > 0) {
        result[i] = s_data[threadIdx.x] - s_data[threadIdx.x-1];
    } else if(i > 0) {
        // I am thread 0 in this block: handle thread block boundary
        result[i] = s_data[threadIdx.x] - input[i-1];
    }
}
__global__ void adj_diff(int *result, int *input) {
    unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;
    __shared__ int s_data[BLOCK_SIZE]; // shared, 1 elt / thread
    // each thread reads 1 device memory elt, stores it in s_data
    s_data[threadIdx.x] = input[i]; // COALESCED ACCESS!

    // avoid race condition: ensure all loads are complete
    __syncthreads();

    if(threadIdx.x > 0) {
        result[i] = s_data[threadIdx.x] - s_data[threadIdx.x-1];
    } else if(i > 0) {
        // I am thread 0 in this block: handle thread block boundary
        result[i] = s_data[threadIdx.x] - input[i-1];
    }
}